Exercise 2: Further Data Manipulation and Visualisation in R

Data, Algorithms and Meaning

Exercises for weeek of

04/04/2018

Further Data Manipulation

Functions

Once we get deeper into writing R code for data manipulation, we soon come across the need to create user-defined functions. We have been using many pre-built functions, such as `str()` and `summary()`, but we can also create our own functions for specific purposes. In R, a function has the general syntax:

```
my_function = function(argument1, argument2) {
  statements
  return(something)
}
```

The function `my_function()` requires two arguments, does something (calculates an object called `something`), then returns `something`. It is important to remember that objects created within the function will not be available in your general R environment unless they are returned. The body of the function is defined within the curly brackets, {}.

Here is a simple example of a function that takes a number and returns the square of it:

```
x_squared = function(x) {
  x_2 = x ^ 2
  return(x_2)
}
```

We can apply this function over a vector of numbers:

```
numbers = 1:10
x_squared(numbers)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

We can also return multiple objects from a function. For example, let's create a function that returns two summary statistics (a.k.a moments of a distribution), plus the number of rows:

```
moments = function(x) {
  # the mean is the first moment of x
  mean = mean(x)
  # standard deviation is the second moment of x
  sd = sd(x)
  # number of rows is given by the length of the vector
  rows = length(x)
  # combine the three vectors into a data frame
  moments = data.frame(mean, sd, rows)
  return(moments)
}
# apply our function to some numbers
moments(numbers)
```

```
##   mean      sd rows
## 1  5.5 3.02765   10
```

Aggregation

Throughout the usual process of preparing data for analysis, we commonly need to do some aggregation. This involves calculting some measures of numeric variables (or even counting rows), aggregating (or grouping) over some factors. For those familiar with Excel, this is the same as using a pivot table. In R we easily do this by using the **`aggregate()`** function.

Let's load the diamonds data set, which you can download from UTS Canvas. Again, to read a csv file in from disk, we can use either the `read.csv()` or `read.table()` functions as follows:

```
data = read.csv("diamonds.csv", header = TRUE)
data = read.table("diamonds.csv", sep = ",", header = TRUE)
```

When you first load a data set into R, it is important to check the structure. It's good to confirm that all the rows and columns we expect have been loaded, but also it is important to check that R has interpreted the data types of our columns correctly. Again, we use the `str()` function:

```
str(data)
```

```
## 'data.frame':    53940 obs. of  11 variables:
##  $ X      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Factor w/ 5 levels "Fair","Good",..: 3 4 2 4 2 5 5 5 1 5 ...
##  $ color  : Factor w/ 7 levels "D","E","F","G",..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Factor w/ 8 levels "I1","IF","SI1",..: 4 3 5 6 4 8 7 3 6 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

In this case, we are looking to see whether our categorical variables have been cast as the `Factor` data type. Commonly numeric variables can be incorrectly set to `Factors` as well. In this case, our only factors are `cut, color` and `clarity` which is correct.

We will start by calculating the average carat for each cut:

```
aggregate(formula = carat ~ cut, data = data, FUN = mean)
```

```
##         cut     carat
## 1      Fair 1.0461366
## 2      Good 0.8491847
## 3     Ideal 0.7028370
## 4   Premium 0.8919549
## 5 Very Good 0.8063814
```

The **`aggregate()`** function requires three key arguments. The first is **formula**, which we saw previously while doing some plotting. This object comes up again and again in all sorts of R functions, most notably with models. It's just a way to specify which variables we want to use, and in what way - you generally need to look up the documentation of the function to see how to specify the formula. In the case of **`aggregate()`**, the variables we wish to be calculated are on the left of the tilda (`~`), while those on the right are grouping factors. The second argument is just our data frame, and the last argument (ironically named as **`FUN`**) specifies

how we want our measures to be calculated. We can use any standard numeric aggregate function (e.g. mean, sd, sum), but can also specify our own functions. (Note: or more on the formula interface see: http://www.dummies.com/programming/r/how-to-use-the-formula-interface-in-r/)

The example above is very simple, so lets go a bit further. We might wish to calculate the mean of several numeric variables. To do this, we need to use the **`cbind()`** function to collect our numeric variables together, then pass this to the formula on the left of the tilda:

```
aggregate(formula = cbind(carat, depth, price, x, y, z) ~ cut, data = data, FUN = mean)
```

```
##         cut     carat    depth    price        x        y        z
## 1      Fair 1.0461366 64.04168 4358.758 6.246894 6.182652 3.982770
## 2      Good 0.8491847 62.36588 3928.864 5.838785 5.850744 3.639507
## 3     Ideal 0.7028370 61.70940 3457.542 5.507451 5.520080 3.401448
## 4   Premium 0.8919549 61.26467 4584.258 5.973887 5.944879 3.647124
## 5 Very Good 0.8063814 61.81828 3981.760 5.740696 5.770026 3.559801
```

We might then wish to add another grouping variable, to see how the means vary over combinations of two variables. In the formula, we simply use **`+`** to add additional grouping variables. Let's try it with **color** and **clarity**. We will also store the result in a data frame, `aggr`, and output the top 10 rows:

```
aggr = aggregate(formula = cbind(carat, depth, price, x, y, z) ~ cut + color + clarity,
                 data = data, FUN = mean)
head(aggr, 10)
```

```
##          cut color clarity     carat    depth    price        x        y
## 1       Fair     D      I1 1.8775000 65.60000 7383.000 7.517500 7.422500
## 2       Good     D      I1 1.0400000 61.35000 3490.750 6.305000 6.293750
## 3      Ideal     D      I1 0.9600000 61.45385 3526.923 6.067692 6.043077
## 4    Premium     D      I1 1.1550000 61.90000 3818.750 6.703333 6.675000
## 5  Very Good     D      I1 0.9500000 62.20000 2622.800 6.244000 6.242000
## 6       Fair     E      I1 0.9688889 65.64444 2095.222 6.170000 6.061111
## 7       Good     E      I1 1.3308696 61.66087 4398.130 6.926522 6.898696
## 8      Ideal     E      I1 1.0377778 61.85000 3559.389 6.346111 6.327222
## 9    Premium     E      I1 1.0430000 60.80667 3199.267 6.393333 6.344333
## 10 Very Good     E      I1 1.0695455 61.48182 3443.545 6.425000 6.436818
##           z
## 1  4.905000
## 2  3.841250
## 3  3.716923
## 4  4.140833
## 5  3.886000
## 6  4.008889
## 7  4.263913
## 8  3.918889
## 9  3.876000
## 10 3.955000
```

If we want to order our data frame by a variable, we can use the **`order()`** function applied to the rows index in the data frame. The first argument is the variable name which we want to sort by, and the second argument is whether we want to sort by decreasing or not. This is handy to remember. E.g., here are the top 6 rows of `aggr` in descending order by `carat`:

```
head(aggr[order(aggr$carat, decreasing = T), ])
```

```
##          cut color clarity    carat    depth    price        x        y
## 31      Fair     J      I1 1.993478 66.46087 5795.043 7.547826 7.457391
## 33     Ideal     J      I1 1.990000 63.50000 9454.000 7.665000 7.605000
## 1       Fair     D      I1 1.877500 65.60000 7383.000 7.517500 7.422500
```

```
## 30 Very Good     I      I1 1.766250 62.10000 6045.125 7.453750 7.452500
## 25 Very Good     H      I1 1.654167 61.81667 5258.833 7.455833 7.437500
## 29   Premium     I      I1 1.605833 61.29167 5044.625 7.343750 7.275000
##           z
## 31 4.990435
## 33 4.875000
## 1  4.905000
## 30 4.628750
## 25 4.600833
## 29 4.482083
```

Lastly, we can use any function we like for the aggregate numeric variable calculation, such as a function similar to the **moments** function we defined earlier. We have to modify the syntax slightly to comply with the FUN argument in aggregate.

```
aggregate(formula = carat ~ cut, data = data,
          FUN = function(x) { c(mean = mean(x), sd = sd(x), rows = length(x)) })
```

```
##         cut   carat.mean     carat.sd   carat.rows
## 1      Fair 1.046137e+00 5.164043e-01 1.610000e+03
## 2      Good 8.491847e-01 4.540544e-01 4.906000e+03
## 3     Ideal 7.028370e-01 4.328763e-01 2.155100e+04
## 4   Premium 8.919549e-01 5.152616e-01 1.379100e+04
## 5 Very Good 8.063814e-01 4.594354e-01 1.208200e+04
```

Note how R preserves the variable's name by adding the calculation name, carat.mean, carat.sd and carat.rows.

Advanced Data Visualisation

We covered some basic data visualisation last week using the built-in graphics library. Now we will look at some more advanced visualisation using ggplot2 and interactive visualisations using **shiny**. We will start with the function qplot in the ggplot2 package.

qplot

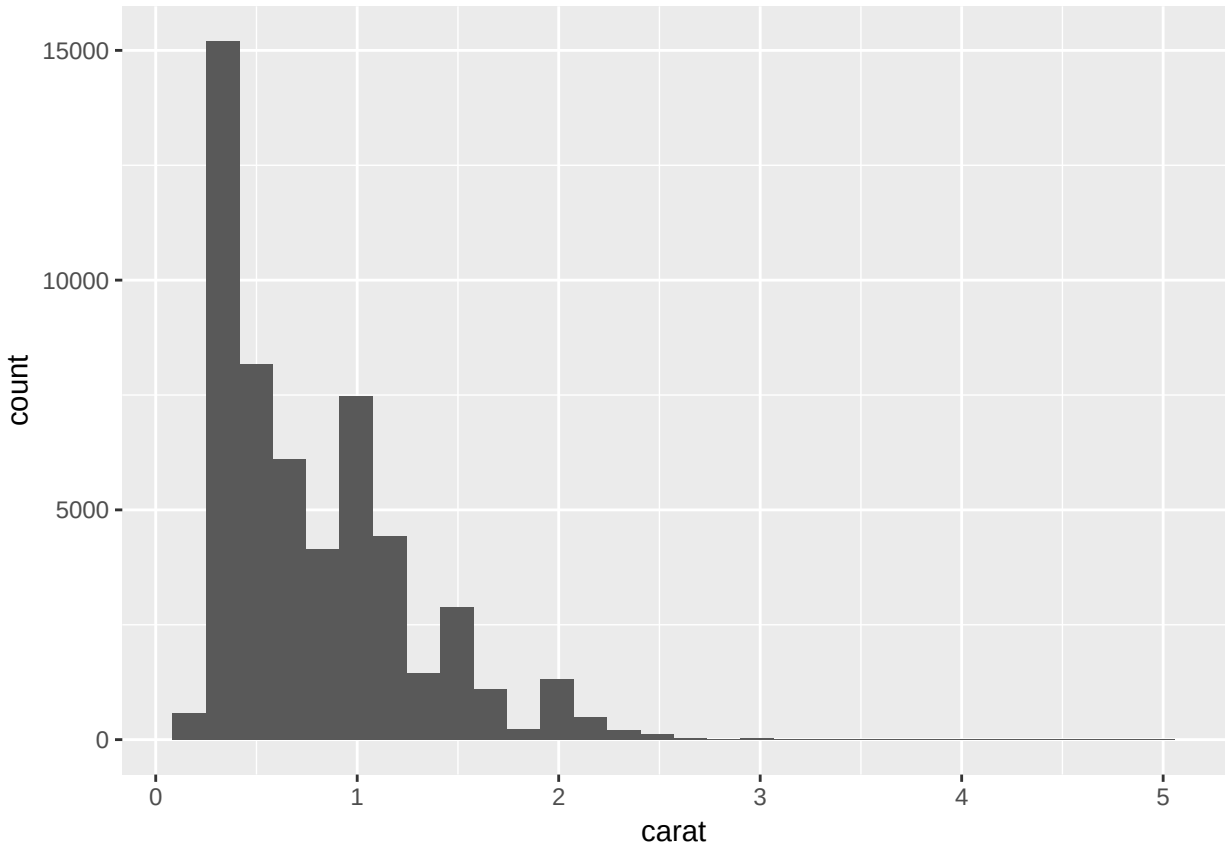
ggplot can produce some very impressive visualisations in R, but it can be difficult to get a grips on the syntax. It is based around what has been called the 'grammar of graphics'. Luckily, there is a lot of material online to help you through this (e.g. http://docs.ggplot2.org/current/). We'll start by using the simplified **qplot()** function (quickplot), which is designed to be a ggplot shortcut with syntax similar to plot. To begin, we load ggplot2 to memory.

```
library(ggplot2)
```

qplot requires three arguments as mandatory, which are the x value (your variable), your data set, and a geom. This last argument stands for geometry, and is how we tell qplot what sort of plot we want. Using the same diamonds data set as before, let's give it the carat variable and see what happens:

```
qplot(x = carat, data = data, geom = "auto")
```

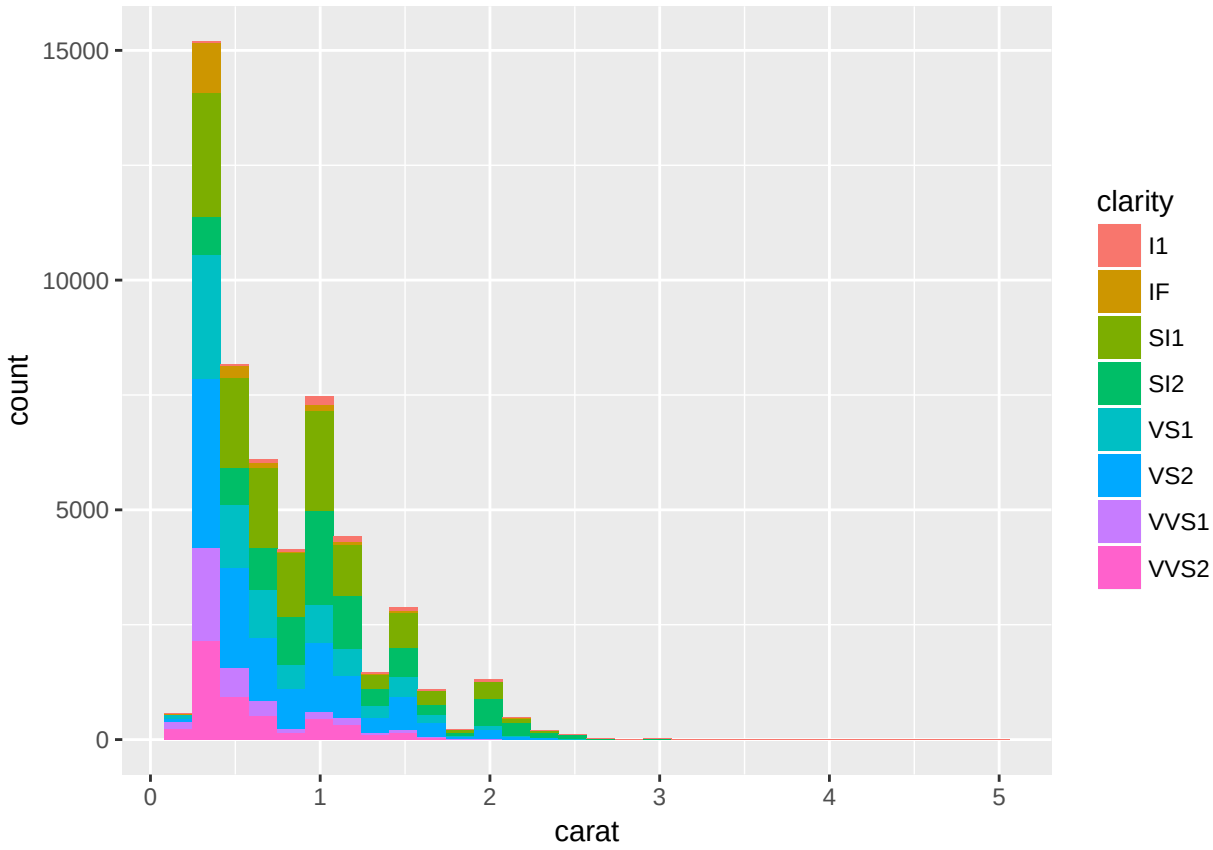
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Success! `qplot` has determined that since `carat` is a numeric variable, it should plot a histogram to show us the distribution. Let's go a bit further and add a `fill` argument to the plot, passing the factor variable `clarity`. This will create a stacked histogram:

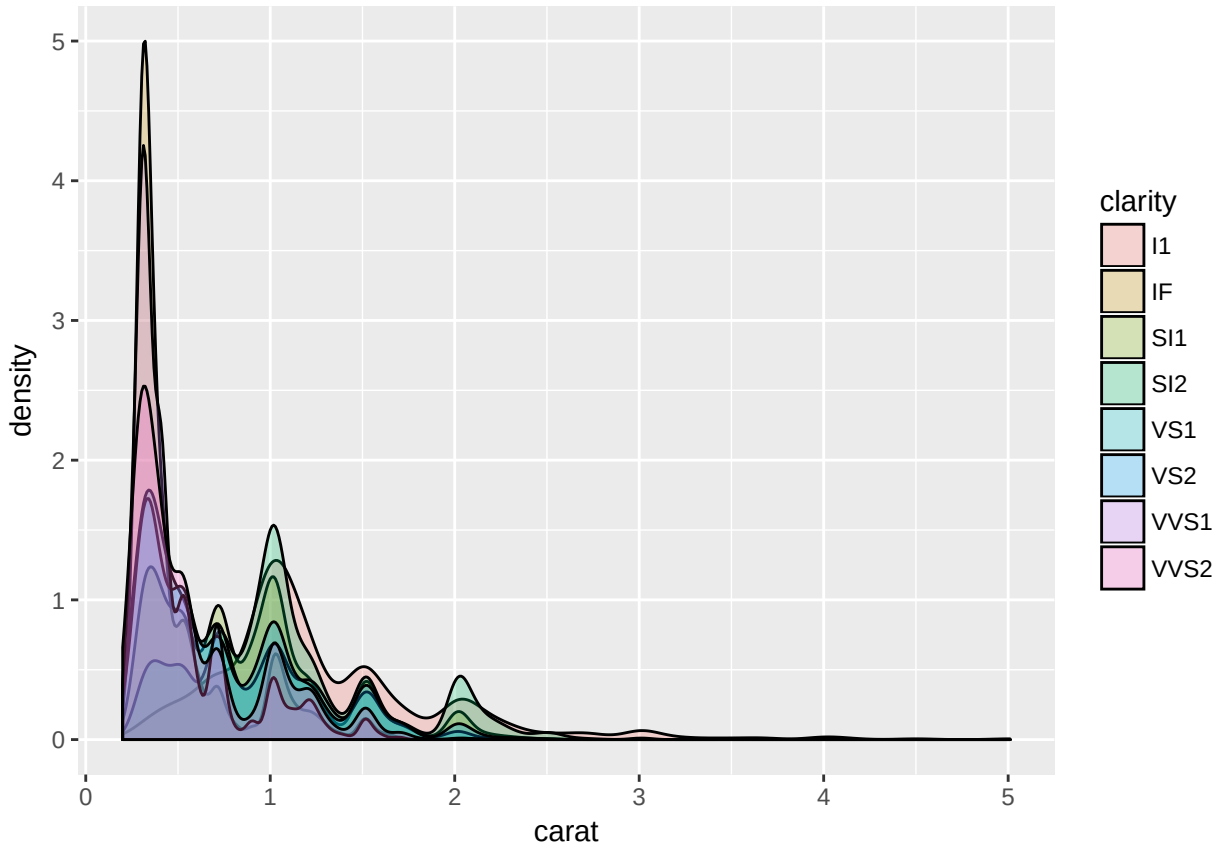
```
qplot(x = carat, data = data, geom = "auto", fill = clarity)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



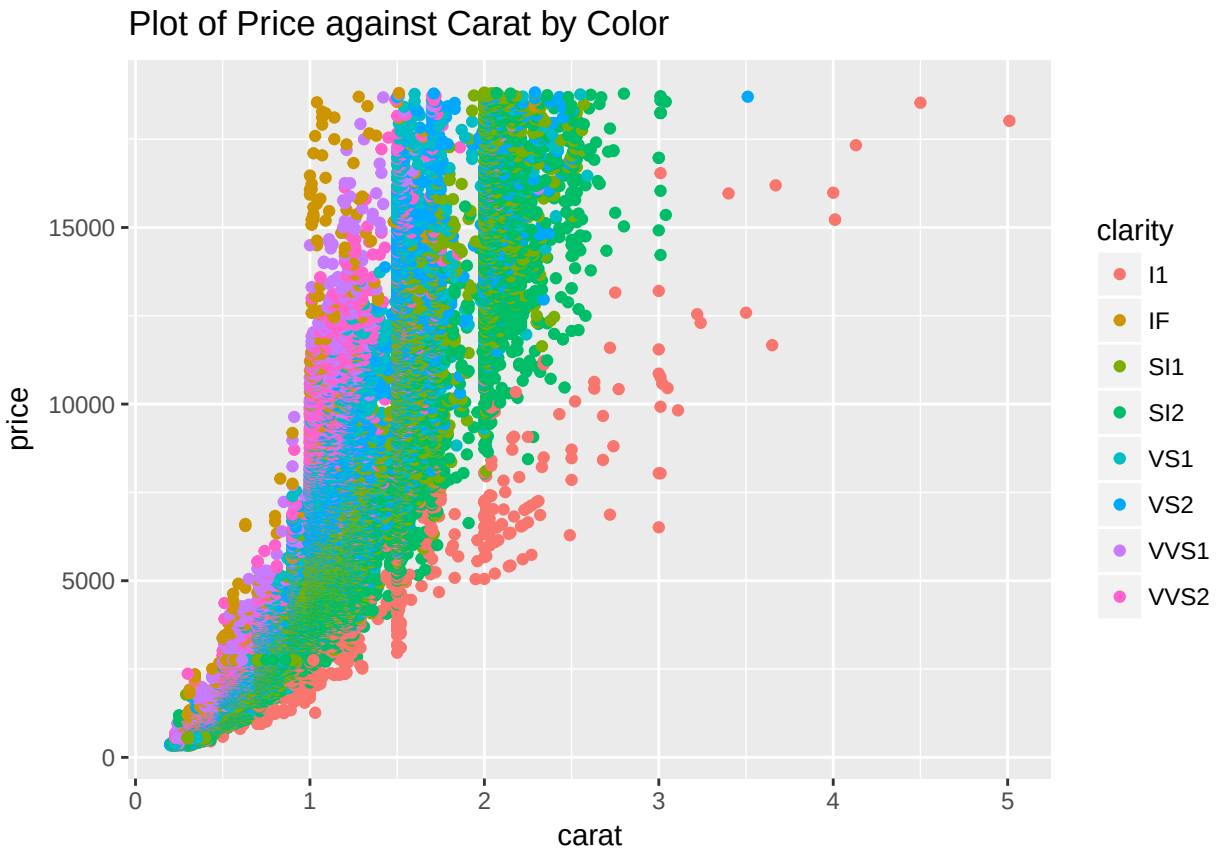
Instead of a histogram, we might want to plot a density distribution. Think of this as a histogram with very narrow bins and a smoothed curve. Density plots tend to show more detail than a histogram, and it is generally easier to compare distributions split by a factor. To create one, we just change the `geom` to be `"density"`.The last parameter, `alpha`, just species how transparent our filled colours are (varies from 0 to 1)

```
qplot(x = carat, data = data, geom = "density", fill = clarity, alpha = I(0.25))
```



Lastly, we might want to look at a scatter plot of `carat` against `price`, coloured by the `color`. We can also add a chart title using the `main` argument. Note that there are over 50,000 data points in this plot:

```
qplot(x = carat, y = price, data = data, geom = "point", color = clarity,
      main = "Plot of Price against Carat by Color")
```



ggplot

`qplot` is great for quickly producing many standard plots using `ggplot`, without knowing the full `ggplot` syntax. Ultimately, if we want to add more customisation and details, we will need to use the core `ggplot` syntax. It can be daunting at first, but learning the core concepts behind the syntax will help. There is a wealth of material online, so don't fret about memorising everything!

Let's start with the core `ggplot` syntax.

```
ggplot(data, aes()) + geom()
```

There are three core elements to any `ggplot` graph. The first is your data set. The second is the **aesthetics**, `aes()`. This encodes what variables in your data set you wish to display, and where you wish to display them. The third component is the `geom()`, the **geometry**. The geometry defines the actual graphical elements displayed in the plot.

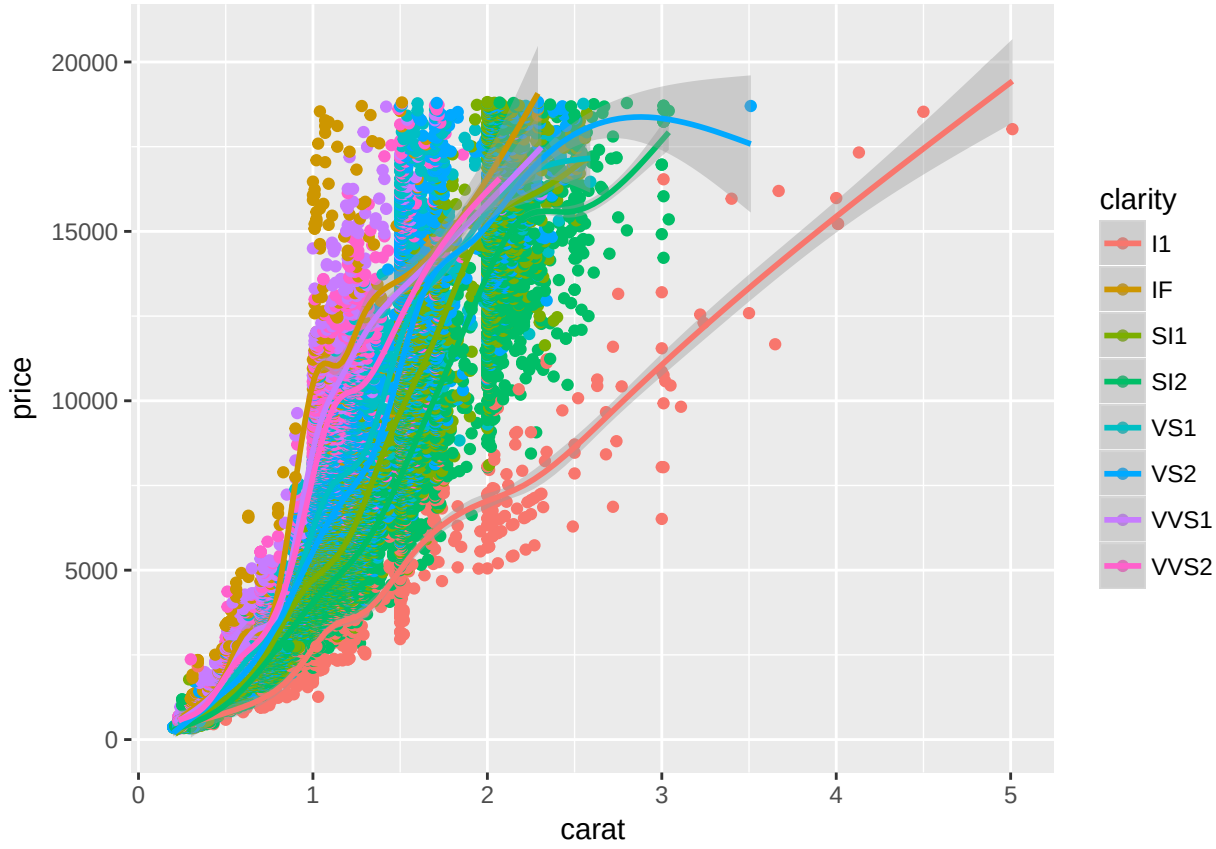
It's easier to understand with an example. The `ggplot` code below reproduces the last `qplot` we created. The `qplot` code is given for reference.

```
ggplot(data = data, aes(x = carat, y = price, color = clarity)) + geom_point()
# qplot code for reference
qplot(x = carat, y = price, data = data, geom = "point", color = clarity)
```

Both lines of code might look similar and definitely perform the same function. However, it is very easy to extend the `ggplot` plot with additional geometries. In the plot, it looks like there might be a difference in the relationship between `price` and `carat` for different `color` values. We can determine if this is the case by overlaying trend lines. We will overlay a trend line geometry using `geom_smooth`, letting R choose a method automatically.

```
ggplot(data = data, aes(x = carat, y = price, color = clarity)) + geom_point() +
  geom_smooth(method = "auto")
```

```
## `geom_smooth()` using method = 'gam'
```

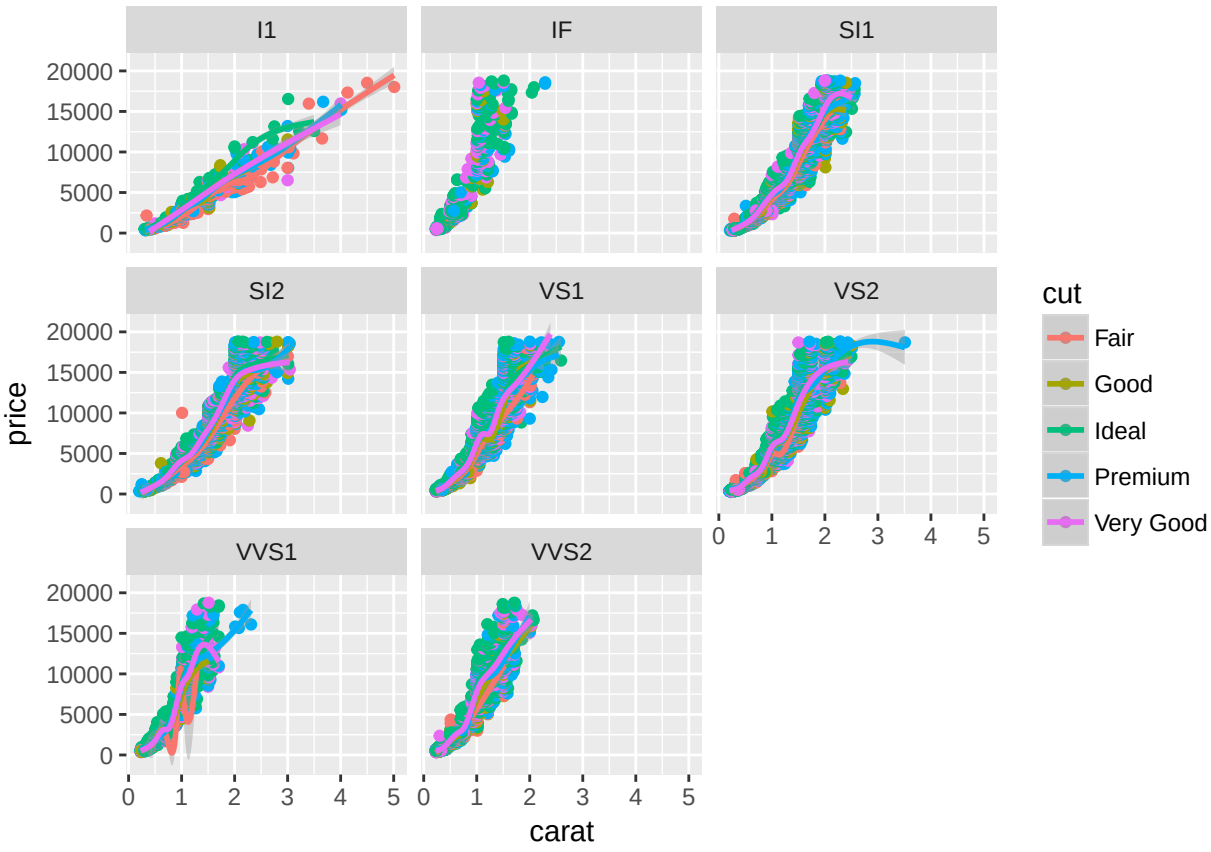


There is a very distinct line for `clarity = I1` which is much less steep then the others. Judging from the plot, it might be worthwhile creating separate plots for each value of `clarity`. This is known as 'faceting' the plot, and we can add the term `facet_wrap( ~ clarity)` to do this. The `wrap` part just tells R to wrap the individual plots over multiple rows to be readable. The argument `~ clarity` tells R that we want to create separate plots for each value of `clarity`. We can facet by additional variables by adding more into the expression. While we're at it, we can also colour the plot by `cut` to see visually if there is any clear effect from the cut type.

```
ggplot(data = data, aes(x = carat, y = price, color = cut)) +
  geom_point() + facet_wrap( ~ clarity) + geom_smooth(method = "auto")
```

```
## `geom_smooth()` using method = 'gam'
```

```
## Warning: Computation failed in `stat_smooth()`:
## x has insufficient unique values to support 10 knots: reduce k.
```



Often while running `ggplot`, R will warn you about certain things. These warnings are usually to do with a failure to produce everything you wanted. While they are not severe enough to create an error and abort the code execution, certain features may not have been produced. In this case, R warns us that `stat_smooth()` failed somewhere. Looking at the plot, we can see that there are no smoothed lines for the `clarity` value `IF`.

Overall, this visualisation is starting to make the relationship between `price`, `carat` and other variables more clear. We'll leave it there with `ggplot`. Mastering this library takes time and patience, and lots of google searches, but is well worth it in the end. For documentation around the range of functions, as well as some additional demonstrations, the following links might be useful:

- http://docs.ggplot2.org/current/
- http://www.statmethods.net/advgraphs/ggplot2.html
- http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html

Shiny

The last section in this exercise involves the `shiny` package. This is purely optional and is included to demonstrate how useful R can be for exploring visual data interactively. `Shiny` is produced by RStudio, who you should be familiar with by now!

`Shiny` basically creates a web-based interactive visualisation using R graphics libraries. It can be viewed as a pop-up in RStudio, or opened directly into a web browser. A `shiny` application can even be hosted on a website, but be sure to check the licensing before doing this. It is a free tool for our current use cases, but there may be a cost if it's deployed commercially.

There are two core components to a `shiny` application, the user interface (UI) and the server. The UI is where the user can interact with the visualisation - interactions can be selected variables, changing parameters,

changing the plot type, filtering, etc. The UI function displays these options in the application, and stores the selected values.

The server function is where the bulk of the work is done. Generally, this will involve creating a visualisation, but sometimes doing some data manipulation as well. Input values selected by the user in the UI are passed to the server function, which can then filter the data set, or modify the output in some way. Have a look at the `shiny` website to see a simple example:

- https://shiny.rstudio.com/

We are now going to build our own `shiny` application on the diamonds data set. We first need to install and load the `shiny` package, and we'll also load `ggplot2` just in case:

```
install.packages("shiny")
```

```
library(shiny)
library(ggplot2)
```

We will use the earlier plots we created using `qplot` as the starting point. However, we allow the user to change the plot geometry and the colouring variable. We will use the histograms and density plots of the variable `price` as the main plot, allowing the user to overlay the three factor variables as colours. Firstly, we will define our UI function.

```
ui = shinyUI(bootstrapPage(

  # selectInput is a function for creating a simple selection.  Display a trend line?
  selectInput(inputId = "variable",
              label = "Select a variable to overlay",
              choices = c("clarity","cut","color"),
              selected = "clarity"),
  # choose the geometry to use
  selectInput(inputId = "geometry",
              label = "Select a geometry",
              choices = c("density", "histogram")),
  # display the plot which we will define in the server
  plotOutput(outputId = "main_plot")
))
```

This is a basic UI function. It allows us to select from three variable choices, `cut`, `color` or `clarity`. The default selected variable is `clarity`. The `inputId` specifies what this input is called in our code, and the `label` is what will appear to the user. The second `selectInput()` function will choose our geometry. The UI is wrapped in the function `shinyUI()` which specifes that this is the UI component of our application. `bootstrapPage()` is another shiny function that specifies the basic layout of our application. Lastly, the `plotOutput()` part is telling R that we will have a plot output called `main_plot` and we want to display it.

Now we will define our server function:

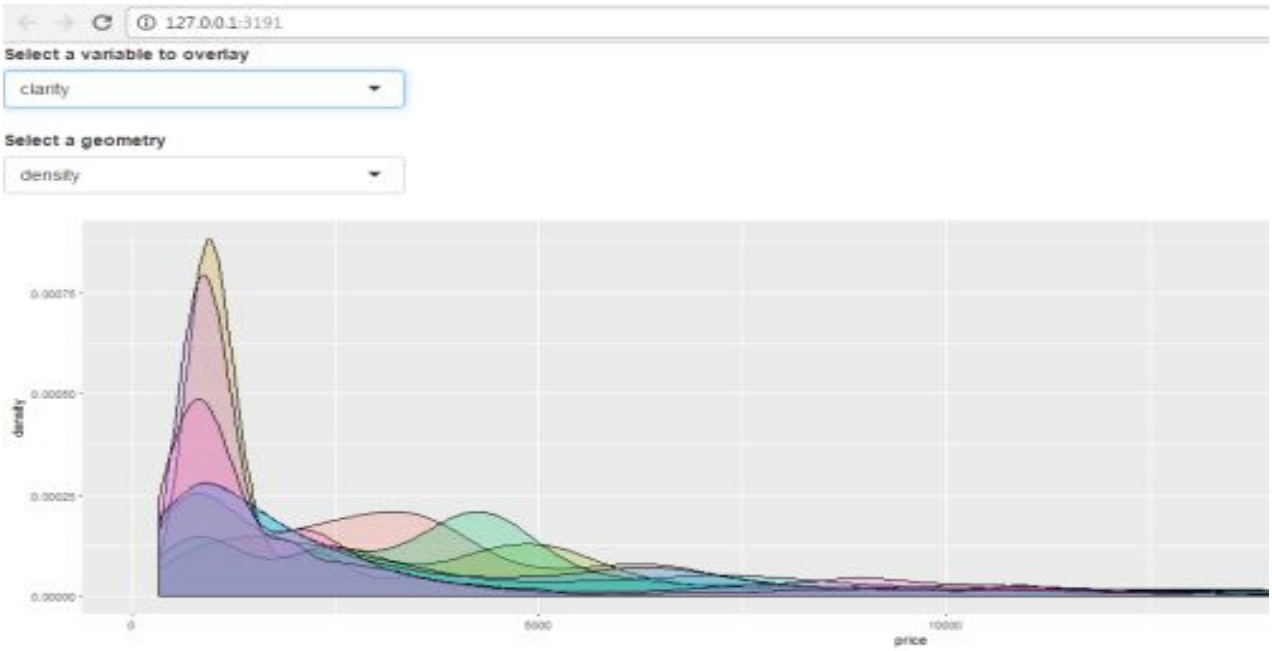
```
server = shinyServer(function(input, output) {
  # function to create plot and pass to UI
  output$main_plot = renderPlot({
    # filter out data frame
    data = data[, c("price", input$variable)]
    # define base plot
    plot = qplot(x = price, data = data, geom = input$geometry, fill = data[, 2], alpha = I(0.25))
    # output the plot
    plot
  })
})
```

The main function `shinyServer` tells R that this is our server, which takes in an `input` object from the UI and passes the `output` object back to the UI. All that is passed as output here is the `main_plot`, created using `renderPlot()`. We first subset our data by the fields `price` and the factor of our choice (`input$variable`), then we produce the plot, selecting the `geom` in line with our choice `input$geometry`.

With our UI and server functions defined, we just need to run the `shinyApp()` function to create our shiny web application. See if it works for you:

```
shinyApp(ui, server)
```

As shown below, this should open up an interactive web page on your default browser.



Experiment with the selections and see how the plot changes. You might want to modify the code in the previous page to display different types of plots.

For much more on Shiny see: https://shiny.rstudio.com/tutorial/